A P2P communication architecture reflecting on the Plugs and Synapses model for ubiquitous computing systems

CHRISTOS GOUMOPOULOS, NIKOS DROSSOS, IOANNIS CALEMIS AND ACHILLES KAMEAS

COMPUTER TECHNOLOGY INSTITUTE, RESEARCH UNIT 3, DESIGN OF AMBIENT INFORMATION SYSTEMS GROUP, PATRAS, GREECE

{GOUMOP, NDROSSOS, CALEMIS, ACHILLES.KAMEAS}@CTI.GR

P.O. BOX 1122, GR-261 10 PATRAS, GREECE

Abstract:

An important requirement for ubiquitous computing systems is the support from communication technologies; P2P networking being the primary candidate. This paper presents a P2P communication architecture that reflects upon a plug/synapse model. The proposed architecture is part of GAS (Gadgetware Architectural Style), a generic architectural style, which allows computational artefacts (eGadgets), to be easily combined into meaningful configurations by users. The plug/synapse model provides the necessary conceptual abstractions to access uniformly eGadget's services and capabilities in order to realize a collective behavior. The implementation of the architecture consists of algorithms and protocols for wireless, connectionless communication as well as mechanisms for internal diffusion of information exchanged.

Keywords: Ubiquitous Computing, P2P Networks, Communication Protocols

Introduction

The main idea behind the “ubiquitous computing” or “ambient computing” or “disappearing computer” concept is that computing services are made available to users throughout their physical environment [1]. The paper describes research that has been carried out in “extrovert-Gadgets”, a research project funded in the context of EU IST/FET proactive initiative “Disappearing Computer” [2].

The e-Gadgets project perspective pushes this idea further, by providing GAS as a conceptual and technological framework that will engage and assist ordinary people in (re) configuring or using systems composed of computationally enabled everyday objects (these objects are sometimes called “artefacts”). In that sense, artefacts are treated as reusable “components” of a dynamically changing everyday environment.

The proposed communication architecture is part of GAS (Gadgetware Architectural Style), a generic architectural style, which can be used to describe everyday environments populated with computational artefacts [3]. A Gadgetware Architectural Style (GAS) constitutes a generic framework shared by both artefact designers and users for consistently describing, using and reasoning about a collection of artefacts.

Basic Concepts

The basic definitions underlying the generic concept are:

- **eGadget**: eGadgets (eGts) are everyday physical objects enhanced with sensing, acting, processing and communication abilities. Moreover, processing may lead to intelligent behavior, which can be manifested at various levels. eGts can be regarded as artefacts that can be used as building blocks to form eGadgetworlds, with the support of GAS.
- **Plugs**: They are software classes that make visible the eGt’s capabilities to people and to other eGts. They may also have tangible manifestation on the eGt’s physical interface, so that users can utilize them in forming synapses.
- **Synapses**: They are associations between two compatible plugs.
- **eGadgetWorld**: An eGadgetWorld (eGW) is a dynamic distinguishable, functional configuration of associated eGts, which communicate and / or collaborate in order to

realize a collective function. eGWs are formed purposefully by an actor (user or other) and appear as functionally unified entities.

Figure 1 illustrates schematically the plug/synapse model in e-Gadgets. Two eGts are connected through a synapse which is established between two plugs forming an eGW.

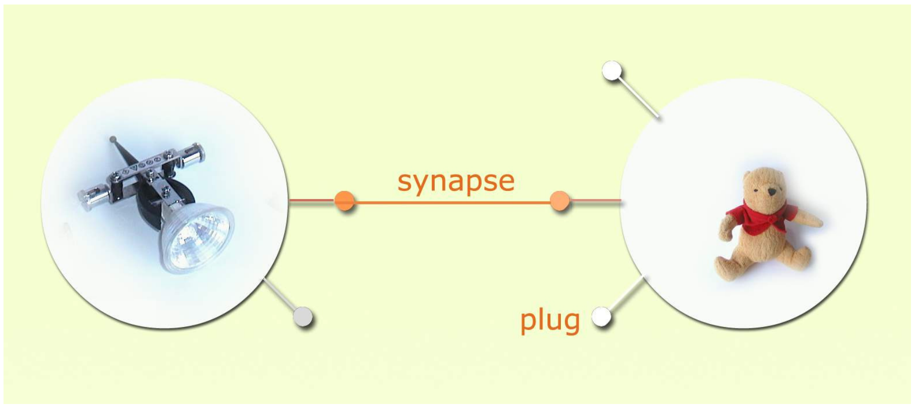

Figure 1: Plug/synapse model in e-Gadgets

Requirements

By nature, eGadgets exhibit a dual presence, both in the real and cyber (digital) worlds. In the former they appear as tangible objects, occupying physical space for a certain amount of time, while in the cyber world, they appear as software entities, which are instantiated and "run" on a processing unit. These two "selves" of an eGt are tightly interrelated: every eGt must have a representation in both worlds and changes in one representation may affect the other. The digital object representations are able to collaborate and allow the user to create meaningful scenarios, while reflecting the results of this collaboration back to the physical world. The architecture and design of a system that would implement the above concepts should satisfy the following requirements:

- Support of ad-hoc peer-to-peer (P2P) networks, since there is no guaranteed infrastructure to rely upon;
- Distributed architecture to meet the needs of an ad-hoc network and increase the robustness and fault tolerance of the system;
- People should not need to be engaged in any type of formal "programming" in order to achieve the desired functions. However, they need to be provided with tools equivalent to a "programmer's workbench", so that they can compose, trace and debug the eGWs;

- eGts are not necessarily provided as all-contained black-box entities, but can also act as "parts" for building larger "wholes". Thus, the principles for deciding on an eGt's level of conceptual granularity need to be identified;
- eGts need to explicitly "advertise" their interconnection capabilities to users through a comprehensible "vocabulary" and metaphors (e.g. by shape, visual probes, handles etc.);
- Fast communication with the hardware, since response times must be minimum, almost real-time;
- Fast communication protocols between the nodes of the system;
- Language independent implementations, since eGts may be built by different manufacturers.

Structure of the paper

The rest of the paper is organized as follows. The next section elaborates on eGts by giving descriptions ranging from a high-level architecture point of view to the component modules comprising the GAS-OS kernel, the software that implements GAS services. The communication module, which is described next implements two P2P schemes. The first one is based on the concepts of plugs and synapses, where as the second one is based on a multi-layer architecture that implements a number of communication protocols serving the needs of the underlying distributed system. Extended functionality can be added to the communication module in terms of providing grouping and routing mechanisms as explained in the end of this section. A discussion of related work follows. The paper closes with the conclusions and a future work outline.

eGadgets

An eGt is an everyday object with an embedded digital board. Its high level architecture is shown in Figure 2. It is composed of a matrix of sensors and actuators, an FPGA-based board which implements communication between the sensor or actuator matrix, and a (processor + memory + wireless) module, which is currently served by an IPAQ or a Laptop.

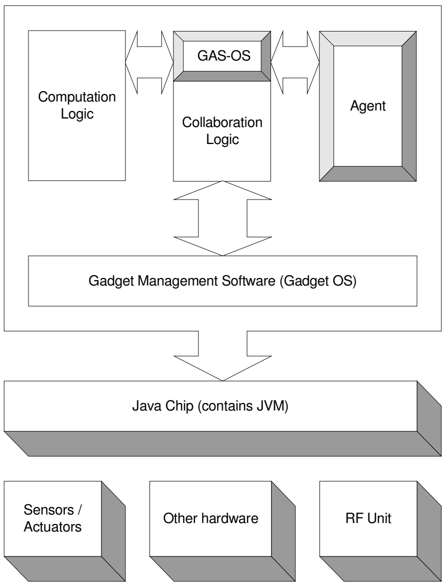

Figure 2: High-level eGadget architecture

The GAS-related middleware includes the following modules:

- The Gadget Management Software (GadgetOS) is responsible for providing access to the eGt resources (e.g. the RF unit, sensors and actuators, etc.) and is implemented per eGt,
- The Collaboration Logic provides service discovery services,
- The Computational Logic implements the intrinsic eGt functions.

The following software modules implement GAS-related services:

- GAS-OS manages the resources of eGts and enables participation in eGWs, through plug and synapse management services,
- Agent implements intelligent mechanisms.

However, eGts may, or may not have all of these components. Moreover, the various components of an eGt (e.g. hardware components) may be implemented by different manufacturers, fact that dictates well-defined interfaces between modules. An intelligent eGt (containing an agent) should have the same GAS-OS as a dumb eGt (not containing an agent). Additionally, accessing the resources of a remote eGt should not be logically different from accessing the resources of the local eGt. A logical access mechanism should hide the implementation of the physical access mechanism. Furthermore, technical constraints like limited memory or CPU capabilities may not permit the full integration of every physical object as an eGt.

The above reasons dictate the specification of a minimal eGt or GAS-OS Kernel that encompasses the minimum number of functionalities that allows the object to obtain a digital self. All the other modules not being necessary for the minimal operation of the eGt, are considered as tools that when present are able to enhance the performance of an eGt and consequently an eGW.

The GAS-OS Kernel encompasses a Communication Module, a Process Manager, a State Variable Manager, a Memory Manager, and an Ontology Manager as shown in Figure 3. In this paper we will be restrained to the presentation only of the Communication Module. For details regarding the other modules of the GAS-OS Kernel the reader is referred to [4].

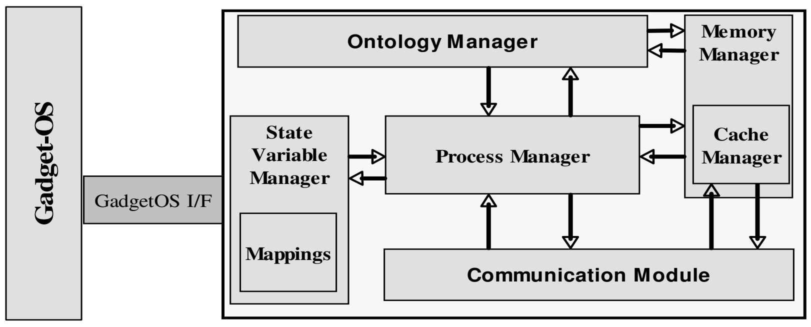

Figure 3: The GAS-OS kernel

The communication module

The communication module is responsible for communication between different GAS-enabled nodes, namely nodes running GAS-OS software. This module implements algorithms and protocols for wireless, connectionless communication as well as mechanisms for internal diffusion of information exchanged.

The inter-eGadget communication can be further divided into plug-to-plug and underlying communication, both realized as peer-to-peer communications. Plug-to-plug is communication in a conceptual level that occurs when two eGts exchange information through a synapse where the type of data is concrete and has a predefined direction. Underlying communication occurs when two eGts exchange information without the existence of a conceptual channel, and this usually happens when they request or grant resources to other eGts, or in general for internal communication of the distributed system.

Groups of GAS-enabled nodes (eGW) that exchange information between each other comprise a peer-to-peer ad-hoc network. The network is characterized as an ad-hoc network because due to the eGt's concept, the network may change dynamically. Those potential changes in the network infrastructure may happen either because of eGts leaving the network, due to portability or failure reasons, or because of new eGts entering the network.

Plugs and Synapses

eGts are allowed to participate in eGWs by using plug and synapse management services provided by the GAS-OS. The plug/synapse model provides the necessary conceptual abstractions to access uniformly eGt's services in order to realize a collective behavior. Figure 4 shows an eGW formed by using plugs and synapses.

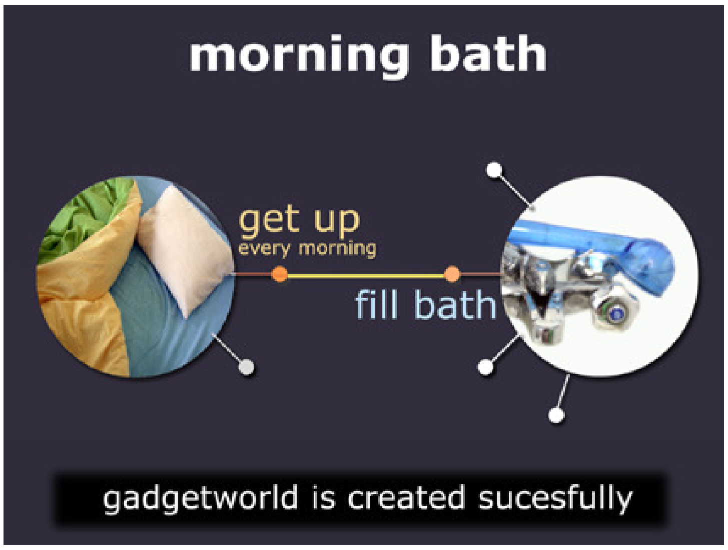

Figure 4: A morning bath eGadgetworld realized as a synapse between plugs

Plugs

A plug belongs to an eGt and is an abstraction of its properties and capabilities. It is the only way other eGts can use an eGt's services and have access to the eGt's properties. Each eGt has several S-Plugs and one T-Plug.

An S-Plug is the plug that describes services and there is one S-Plug per service. S-Plugs have the following attributes:

- *eGadget Handle*: A reference to an eGt wrapper that specifies all the GAS-OS specific attributes of a component. Every eGt owns an eGt Handle that allows the embodiment of the eGt to the GAS-OS.
- *Name*: The name of the plug must be unique for the eGt. On the other hand different eGts can have plugs with the same name.
- *Type*: The type of the plug refers to its capability to accept or send data or both. Thus we distinguish Input, Output and Input-Output plugs.
- *Data type*: The data type specifies the kind of data a plug can accept or send. At this stage data types are limited to primitive data types, such as Integers, Strings and Booleans. However, future versions of GAS-OS will support more data types, even custom ones.

Depending on whether the plug is an input or output one, we distinguish the following cases:

- An output plug carries a shared object. Shared objects represent the actual data an output plug sends to other plugs. When shared object values are altered by their owner eGt, they implement an internal mechanism to notify all the listener plugs for this change.
- An input plug carries a shared object listener. Shared object listeners listen for changes in shared objects of the provider plug and notified when something has changed.
- Obviously if a plug is both input and output it owns both a shared object and a shared object listener.

S-Plugs also provide connection and disconnection listeners that are called initially when a synapse is achieved and allow some initialization to take place.

Every eGt provides also a TPlug, which can be queried to provide the following kind of information: the state the eGt's ID, the list of its Plugs and current Synapses, the list of its physical properties.

Synapses

A synapse is a connection between two S-Plugs. Synapses are specified by the user either via direct manipulation of the eGts GUI editor or via specially developed tools [5]. The synapsing process is as follows:

- Synapse Request: One of the eGts sends a Connection request message to the other eGt. The message contains information concerning the local and remote plug.
- Synapse Response: When the target eGt receives the message and checks the plug compatibility of the remote and local plugs. If plug compatibility test is passed, an instance of the remote plug is created and a positive response is sent to the eGt that initiated the synapse. The instance of the remote plug is notified for changes by its remote counterpart plug and in fact serves as an intermediary communication level. In case of a negative plug compatibility test, a negative response message is sent to the remote plug, while no instance of the remote plug is created.

When the initial eGt receives a positive response, it also creates an instance of the remote plug, for the same reason as before, and the connection is established.

Multi-layer Architecture

The communication module was designed having in mind the basic principles and definitions of the Jxta project [6] with respect to the architecture of the GAS-OS. It is a decentralized (requiring no fixed infrastructure or the support of any other entity except computing peers) messaging based system abstracting the underlying network and communication protocol and providing services through a well-defined interface. All communication between peers and even within the platform layers takes place via asynchronous XML based messages. The design and architecture of the communication module is based on eComp [7] a previous attempt to build a custom P2P architecture. The communication system is divided into three separate layers, each one providing a different level of abstraction for the communication process.

- *Peer*. A peer is any networked device that implements one or more communication protocols. Each peer operates independently and asynchronously from all other peers and is uniquely identified by a Peer ID. Peers allow for other components higher in the hierarchy, like the process manager, to register themselves as event listeners, in order to receive notifications for incoming data. Peers are also responsible for the implementation of communication protocols, such as discovery protocol, message exchanging protocols, routing protocol etc.

- *Pipe*. Each peer owns two pipes: an Input pipe waiting for incoming messages and an Output pipe that sends either unicast or multicast messages to the network. Pipes are an asynchronous and unidirectional message transfer mechanism used for service communication. Pipes are indiscriminate; they support the transfer of any object, including binary code, data strings, and Java technology-based objects. However, as GAS-OS aims to be a platform and language independent system, only XML messages are transferred through pipes.
- *Endpoint*. Each pipe is bound to one or two Endpoints, which are considered as the fundamental networking units of the communication module. The Endpoint may be associated to specific network resources such as a TCP port and associated IP address. An input pipe owns two Endpoints, one for listening multicast messages and one for listening TCP P2P messages. An output pipe owns one Endpoint which is configured dynamically to send either multicast messages to a default multicast group, or P2P TCP messages.

Figure 5 demonstrates the communication process to and from another GAS-enabled node, using the communication module described.

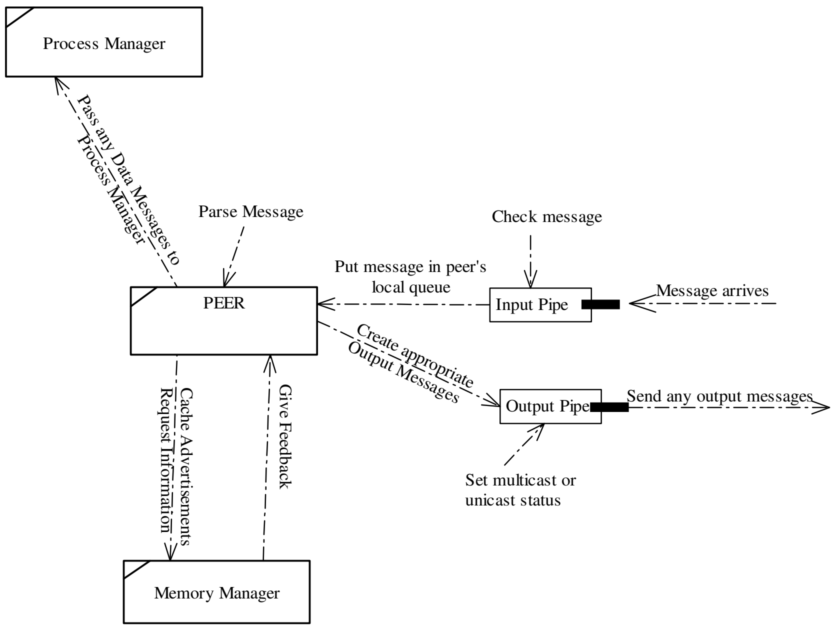
Process Manager
Pass any Data Messages to Process Manager
Parse Message
Check message
Put message in peer's local queue
PEER
Input Pipe
Message arrives
Create appropriate Output Messages
Output Pipe
Send any output messages
Set multicast or unicast status
Cache Advertisements Request Information
Give Feedback
Memory Manager

Figure 5: Incoming /outgoing messages

Protocols

The GAS-OS communication module defines a series of XML message formats, and protocols, for communication between eGts, which use these protocols to discover each other, advertise and discover network resources, and communicate and route messages. There are currently four communication protocols:

- *Gadget Discovery Protocol (GDP).* This protocol is used to discover eGts in the network using resources as criteria. Resources are represented as advertisements, and are usually information about a remote eGt (i.e. ID, Services, Relay, etc.). This information can then be used to establish a P2P communication. An eGt creates a discovery message that is sent to all listening eGts through a Multicast Endpoint. There are two kinds of discovery messages, the simple ones, where a specific eGt ID is requested and the composite ones, where either a specific service from an unknown eGt is requested or the whole world. An optional caching mechanism may improve the performance of the protocol by maintaining tracking information for remote peers that has been communicated in the past, so that previous experience is well utilized, thus saving resources and effort.

- *Gadget Advertisement Protocol (GAP).* This protocol is used to advertise one or more properties of an eGt. An advertisement message is usually a response to a discovery message, thus formed according to the requested information.
- *Gadget Information Protocol (GIP).* This protocol is used for exchange of data between two eGts. It uses a single unicast pipe to send the information to a specific peer. Each data message is folded inside an envelope that contains the recipient's identification.
- *Message Acknowledging Protocol (MAP).* This underlying protocol is responsible for the robustness of the communication system. Its main task is to provide acknowledgements (ACK) for each arrived message to the sender and NACK receipts for each lost message. Furthermore if a message is lost the MAP protocol is responsible for resending the message to the target eGt.

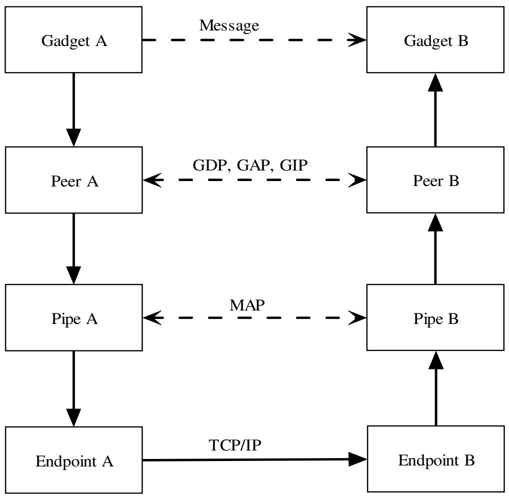

Figure 6: Communication protocols stack

Figure 6 summarizes the communication process between eGts A and B using the above scheme.

Extended functionality

In order to extend the functionality and efficiency of GAS-OS, regarding the communication module, one could augment GAS-OS with grouping and routing capabilities.

Grouping

The purpose for using peer groups, as in [8], is to reduce network load (messages) throughout the network. The networking approach without peer groups may become excessively resource consuming, in terms of network bandwidth and power consumption when dealing with a large number of eGts, since service discovery, for example, may become too expensive. Peer groups introduce a new network topology known as “Federated”. Federated architectures [9] introduce a new notion for peer-to-peer systems by combining the architecture of centralized systems into decentralized ones. Most peers have a centralized relationship to a server-like node or “yellow pages” forwarding all queries to this node which essentially becomes a group leader. But instead of being standalone servers, these nodes may band themselves together in a Gnutella-like decentralized network, propagating the relevant queries.

Grouping, in the case of the service discovery example, is essentially a mechanism to control the amount and availability of service/resource advertisements and to promote scalability limiting the network coverage of a request and the number of reply messages. Every node in the P2P network can belong to one or more groups (alternatively scopes). Server nodes store only services advertised having a scope domain matching the domain of their belonging. Client nodes send unicast request messages only to servers belonging to the same scope. There must be, consequently, a way for the networked nodes to learn their group(s). An example may be by reading a configuration file. The members of a group could be defined based on semantic information, for example common capabilities, or could be defined based on the vicinity of nodes, for example existing in the same room or building.

A group leader, who can be any node inside the group, but usually one, that has higher capabilities than the others, manages peer groups. Inter-group communication is done between group leaders of different groups. Therefore, for resource discovery, a peer just needs to send the query message to the group leader, who is then responsible for discovering the target resource. Query messages are formed provided that there is a mechanism which uniquely identifies the searched resource. We assume that each peer will have a universally unique ID that is hard coded into the device. Resource IDs then could be derived uniquely from the peer ID. In this ID system a peer has to track only the resources it creates and the IDs it assigns to make up new unique IDs. In a more elaborate scheme the resource IDs could be derived from the specifications of the ontology pervading a specific eGt.

In addition, there are several issues that must be considered in order to use the peer group policy effectively in eGts:

- Grouping must not be necessary for the operation of the GAS-OS, but should rather be considered as a tool that will just enhance performance; a mixed scheme should therefore be provided;
- Grouping may not always improve the network's performance. A bandwidth usage analysis that we have made for a resource discovery protocol showed that in configurations with less than 17 eGts, acting either as resource consumers or resource providers, the use of a group leader would be an overhead;
- Problems like leader election, in case of failures, are raised;
- Having to deal with wireless networks whose structure changes dynamically and frequently, what happens when a new peer enters or leaves the group or if a peer does not find the leader, is another issue;
- Finally, a critical question is on which criteria the decision for constructing a peer group will be based on.

Routing

Another major issue is what an eGt should do when it cannot reach a synapse-peer directly. A solution is to apply the relay technique [10]. Suppose that peer B is outside peer A's communication range. Peer D on the other hand, is aware of peer B's location and is capable of communicating with both A and B. In order to establish communication between peers A and B we use peer D as a facilitator. This technique improves the credibility of the network and increases the maximum affordable distance between two peers. Two questions raised at this point however, are how a peer is set as a relay, and what is the maximum number of relay hops required in order to find the target peer. Several known routing techniques can be employed: Pro-active, Re-active, Dynamic Cluster-based and hybrid, and some of the most popular algorithms are: Optimized Link-State Routing (OLSR), Ad-hoc On-Demand Distance Vector (AODV), Temporally-Ordered Routing Algorithm (TORA) and Zone Routing Protocol (ZRP). Having in mind the restrictions posed by the eGts project, re-active techniques seem a rather suitable solution, as no image of the network is required inside the nodes. When a node wants to reach one other, it searches the route on its own initiative, maintaining the traversed paths after routing is accomplished. The use of existing algorithms like TORA, which is the most popular, is under consideration for future versions of GAS-OS.

Related Work

During the design process of GAS-OS several existing solutions for the communication module were studied in terms of suitability to our concept and requirements. An attractive approach is represented by SOAP [11], a lightweight protocol for exchange of information in a decentralized, distributed environment that provides an XML messaging system for the communication scheme. Nevertheless, the specification given for creating tags for the message envelopes, encoding rules and RPC representation are rather complex and create large messages, which may reduce the network performance when scalability issues occur.

In addition, pure Peer-to-Peer implementations, such as JXTA [6], provide a simple and clear mechanism for ad-hoc systems. The currently available version, though, has been developed with the Java Standard edition in mind. Regardless of the fact that ports to other Java editions and implementations on other platforms are available, most of them lose some functionality. Jxme, the port of Jxta [12] to Java Micro Edition [13], is a bare-naked implementation and, above that, it requires the services of a Jxta relay to function as a peer in a P2P network.

Finally, event–based publish/subscribe systems seem an alternative option. Such systems use events as their basic communication mechanism; subscribers express their interest in receiving certain events, while publishers publish events that will be delivered to all interested event subscribers. Hermes [14] is a novel architecture that unites the areas of middleware systems and publish/subscribe communication to provide an event-based middleware. Publish/subscribe architectures though, use a central event dispatcher to record all publications in the system and notify the corresponding subscriber. Consequently, this implies a specific infrastructure as well as lack of autonomy, acceptable factors considering that the application domain of Hermes is e-commerce and business applications over the Internet.

Conclusions and Future Directions

In this paper we have presented the communication module of GAS-OS the software that implements GAS (Gadgetware Architectural Style), a generic architectural style, which can be used to describe everyday environments populated with computational artefacts. The proposed communication architecture implements two P2P schemes. The first one reflects upon a plug/synapse model, which provides the necessary conceptual abstractions to access uniformly artefact's services and capabilities in order to realize a collective behavior. The second one is based on a multi-layer architecture that implements a number of communication protocols serving the needs of the underlying distributed system. Extended functionality and improved performance can be gained by incorporating mechanisms for grouping peers and introducing routing protocols into the communication module.

In our near future plans is the implementation of an extended resource discovery protocol that will be based on a semantic rich query language to support attributes that can be specified in the core ontology of the GAS-OS. In that way, services and resources that are required by an eGt can be requested in a uniform way. The protocol will support two alternative architectures. The first one is the current scheme of resource providers and resource consumers communicating directly with each other with the optional and undeterministic use of resource cache within each eGt. With the aim of reducing traffic, a second architecture will be introduced involving one or more resource directories acting as databanks to which resource providers can register resources and resource consumers can request resources.

Currently, several studies have been conducted including a formal specification of the reactive eGts, studies on the scalability of the concept regarding wireless networking, resource discovery, sensor arrays and multiple agents. Especially, scaling issues will be studied further by conducting simulations, for instance using the NS2 tool. Finally, the issue of providing more complete specification of plugs and their QoS attributes and requirements will be examined. In this context, non-functional specifications will be included in the description of Plugs and Synapses.

References:

1. Weiser M. Some Computer Science Issues in Ubiquitous Computing, *Communications of the ACM*, 36(7), pp 75-84, July 1993

2. The Disappearing Computer initiative: http://www.disappearing-computer.net/

3. Kameas A. et al, *An Architecture that Treats Everyday Objects as Communicating Tangible Components*, Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom03), Fort Worth, USA, March 2003

4. Kameas A. et al, D9 - Specifications and design of artefacts v2.0, eGadgets IST-2000-25240 Technical Report, March 2003

5. Mavrommati I., Kameas A. *End user programming tools in ubiquitous computing applications*, 10th International Conference on Human - Computer Interaction, (HCI International), June 2003, Crete, Greece

6. Project JXTA website: http://www.jxta.org

7. Kameas A. et al, *eComp: An Architecture that Supports P2P Networking Among Ubiquitous Computing Devices*, Proceedings of the 2nd International Conference on Peer-to-Peer Computing (P2P'02), Linköping, Sweden, Sept 2002

8. Guttman E. Service Location Protocol: Automatic Discovery of IP Network Ser-vices, *IEEE Internet Computing*, 3(4): 71-80, 1999. http://computer.org/internet/

9. Minar N. Distributed Systems Topologies: Part 1 and 2, August 2002 http://www.openp2p.com/pub/a/p2p/2002/01/08/p2p_topologies_pt2.html

10. Celebi E. *Performance Evaluation of Wireless Mobile Ad Hoc Network Routing Protocols*, MS Thesis, 2001, http://www.cmpe.boun.edu.tr/emre/research/msthesis/

11. SOAP website: http://www.w3.org/TR/SOAP

12. Arora C., Haywood K., Pabla S. JXTA for J2ME™ – Extending the Reach of Wireless with JXTA Technology, March 2002, http://www.jxta.org/project/www/docs/JXTA4J2ME.pdf

13. J2ME website: http://java.sun.com/j2me/

14. Pietzuch P.R., Bacon J.M. *Hermes: A Distributed Event-Based Middleware Architecture*, 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02), Vienna, Austria